本科生实验报告

实验课程：操作系统原理实验

实验名称：编译内核/利用已有内核创建 OS

专业名称：计算机科学与技术

学生姓名：张玉瑶

学生学号：23336316

实验地点：实验楼 B203

实验成绩：

报告时间：2025 年 4 月 28 日

Section 1 实验概述

- 实验任务 1：学习可变参数机制，然后实现 printf，做出改进。
- 实验任务 2：自行设计 PCB，可以添加更多的属性，如优先级等，然后根据你的 PCB 来实现线程，演示执行结果。
- 实验任务 3：编写若干个线程函数，使用 gdb 跟踪 c_time_interrupt_handler、asm_switch_thread 等函数，观察线程切换前后栈、寄存器、PC 等变化，结合 gdb、材料中“线程的调度”的内容来跟踪并说明下面两个过程。

  1.一个新创建的线程是如何被调度然后开始执行的。

  2.一个正在执行的线程是如何被中断然后被换下处理器的，以及换上处理机后又是如何从被中断点开始执行的。
- 实验任务 4：实现自己的调度算法。

Section 2 实验步骤与实验结果

---------- 实验任务 1：printf 的实现----------

- 任务要求：学习可变参数机制，然后实现 printf，做出改进。
- 思路分析： 通过可变参数宏（va_list）访问栈中的参数，结合格式化字符串解析，按需输出字符。具体步骤包括：

  1) 定义 va_start、va_arg、va_end 宏，按 4 字节对齐从栈中提取参数；

  2) 遍历格式字符串，普通字符直接缓存，遇到%则解析类型（如%d、%x），调用 itos 将数字转为字符串；

  3) 使用缓冲区暂存结果，满时触发底层输出函数（STDIO::print），最终返回总字符数。
- 实验步骤：

  1.定义一个具有可变参数的函数 `print_any_number_of_integers( int n,…);` ，先要了解到其中的:

  1) va_list(parameter)是一个可以指向可变参数的指针；
  2) va_start(parameter,n)是利用固定列表最后一项 n 来锁定地址、令 parameter 指向可变参数第一项；
  3) va_arg(parameter,int)将 parameter 所指的变量以 int 输出，同时指向下

一个可变参数;

4) va_end(parameter)清空指针。

再了解下列四个宏的具体实现是什么样的：

1）_INTSIZEOF(n) 返回的是 n 的大小进行 4 字节对齐的结果。注意到，4 的倍数在二进制表示中的低 2 位是 0，而任何地址和 0xfffffffc(~(sizeof(int)-1))相与后得到的数的低 2 位为 0，也就是 4 的倍数，即相当于上面公式除 4 再乘以 4 的过程。但是，直接拿一个数和 0xfffffffc 相与得到的结果是向下 4 字节对齐的，为了实现向上对齐，我们需要先加上(sizeof(int)-1)后再和 0xfffffffc 相与，此时得到的结果就是向上 4 字节对齐的。

2）va_start(ap,v)的逻辑 ap＝(va_list)&v 是把指向可变参数列表的指针 ap 指向一个地址，并且转化为 va_list 类型，这个地址是由最后一个固定参数的地址加上它的大小+ _INTSIZEOF(v)得来的。

3）va_arg 的作用：(ap += _INTSIZEOF(type))先令 ap 指向下一个可变参数，((ap += _INTSIZEOF(type)) - _INTSIZEOF(type))意思是把已经得到的下一个地址再减回刚刚加上的大小，得到原来的地址，最后在前面加上(type *)进行类型转换，意思是形成一个 type 类型的指向当前可变参数的指针，最后取指针输出。

4）va_list 将 ap 指向 0，清空指针。

```
typedef char *va_list;
#define _INTSIZEOF(n) ((sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1))
#define va_start(ap, v) (ap = (va_list)&v + _INTSIZEOF(v))
#define va_arg(ap, type) (*(type *)((ap += _INTSIZEOF(type)) - _INTSIZEOF(type)))
#define va_end(ap) (ap = (va_list)0)

void print_any_number_of_integers(int n, ...)
{
    // 定义一个指向可变参数的指针parameter
    va_list parameter;
    // 使用固定参数列表的最后一个参数来初始化parameter
    // parameter指向可变参数列表的第一个参数
    va_start(parameter, n);

    for ( int i = 0; i < n; ++i ) {
        // 引用parameter指向的int参数，并使parameter指向下一个参数
        std::cout << va_arg(parameter, int) << " ";
    }

    // 清零parameter
    va_end(parameter);

    std::cout << std::endl;
}
```

2．实现 printf。

1）先设置缓冲区。当字符串过大时会超过函数调用栈的大小。所以我们需要定义一个缓冲区，然后对 fmt 进行逐字符地解析，将结果逐字符的放到缓冲区中。放入一个字符后，我们会检查缓冲区，如果缓冲区已满，则将其输出，然后清空缓冲区，否则不做处理。以下是 buffer 的逻辑。

```
int printf_add_to_buffer(char *buffer, char c, int &amp;idx, const int
BUF_LEN)
{
    int counter = 0;

    buffer[idx] = c;
    ++idx;

    if (idx == BUF_LEN)
    {
        buffer[idx] = '\0';
        counter = stdio.print(buffer);
        idx = 0;
    }

    return counter;
}
```

2）实现格式化输出。printf 首先找到 fmt 中的形如%c,%d,%x,%s 对应的参数，然后用这些参数具体的值来替换%c,%d,%x,%s 等，得到一个新的格式化输出字符串，这个过程称为 fmt 的解析。以下在原有的基础上增加实现了“%f”，“%.nf”。

```
int printf(const char *const fmt, ...)
{
    const int BUF_LEN = 32;
    int temp;
    char buffer[BUF_LEN + 1];
    char number[33];
    int precision;
     int digit;
     int int_part;
    int idx, counter;
    double fractional_part;
    double tempp;
    va_list ap;

    va_start(ap, fmt);
    idx = 0;
    counter = 0;

    for (int i = 0; fmt[i]; ++i)
    {
        if (fmt[i] != '%')
        {
```

```
            counter += printf_add_to_buffer(buffer, fmt[i], idx, BUF_LEN);
        }
        else
        {
            i++;
            if (fmt[i] == '\0')
            {
                break;
            }

            switch (fmt[i])
            {
            case '%':
                counter  +=  printf_add_to_buffer(buffer,  fmt[i],  idx,
BUF_LEN);
                break;

            case 'c':
                counter += printf_add_to_buffer(buffer, va_arg(ap, char),
idx, BUF_LEN);
                break;

            case 's':
                buffer[idx] = '\0';
                idx = 0;
                counter += stdio.print(buffer);
                counter += stdio.print(va_arg(ap, const char *));
                break;

            case 'd':
            case 'x':
                temp = va_arg(ap, int);

                if (temp &amp;lt; 0 &amp;amp;&amp;amp; fmt[i] == 'd')
                {
                    counter  +=  printf_add_to_buffer(buffer,  '-',  idx,
BUF_LEN);
                    temp = -temp;
                }

                itos(number, temp, (fmt[i] == 'd' ? 10 : 16));

                for (int j = 0; number[j]; ++j)
                {
                    counter  +=  printf_add_to_buffer(buffer,  number[j],
idx, BUF_LEN);
                }
                break;
                            case 'f':
                tempp = va_arg(ap, double);
                precision = 6;


                 if (tempp &amp;lt; 0)
                 {
                       counter  +=  printf_add_to_buffer(buffer,  '-',  idx,
BUF_LEN);
                       tempp = -temp;
```

```
                 }

                int_part = (int)tempp;
               itos(number, int_part, 10);

                for (int j = 0; number[j]; ++j)
                {
                    counter  +=  printf_add_to_buffer(buffer,  number[j],
idx, BUF_LEN);
                }


                counter   +=   printf_add_to_buffer(buffer,   '.',   idx,
BUF_LEN);


                fractional_part = tempp - int_part;
                for (int j = 0; j &lt; precision; ++j)
                {
                    fractional_part *= 10;
                    digit = (int)fractional_part;
                    counter += printf_add_to_buffer(buffer, '0' + digit,
idx, BUF_LEN);
                    fractional_part -= digit;
                }
                break;

           case '.'://处理点 nf
               int np = fmt[++i]-'0';

               if(fmt[++i]!='f') break;

               tempp = va_arg(ap, double);
               precision = np;


                if (tempp &lt; 0)
                {
                     counter  +=  printf_add_to_buffer(buffer,  '-',  idx,
BUF_LEN);
                     tempp = -tempp;
                 }
                int_part = (int)tempp;
                itos(number, int_part, 10);


                for (int j = 0; number[j]; ++j)
                {
                    counter  +=  printf_add_to_buffer(buffer,  number[j],
idx, BUF_LEN);
                }


                counter   +=   printf_add_to_buffer(buffer,   '.',   idx,
BUF_LEN);


               fractional_part = tempp - int_part;
```

```
                for (int j = 0; j &lt; precision; ++j)
                {
                    fractional_part *= 10;
                    digit = (int)fractional_part;
                    counter += printf_add_to_buffer(buffer, '0' + digit,
idx, BUF_LEN);
                    fractional_part -= digit;
                }
                break;
            }
        }
    }

    buffer[idx] = '\0';
    counter += stdio.print(buffer);

    return counter;
}
```

其中对于%d 和%x，我们需要将数字转换为对应的字符串。一个数字向任意进制表示的字符串的转换函数 itos（）如下所示。

```
void itos(char *numStr, uint32 num, uint32 mod) {
    // 只能转换 2~26 进制的整数
    if (mod < 2 || mod > 26 || num < 0) {
        return;
    }

    uint32 length, temp;

    // 进制转换
    length = 0;
    while(num) {
        temp = num % mod;
        num /= mod;
        numStr[length] = temp > 9 ? temp - 10 + 'A' : temp + '0';
        ++length;
    }

    // 特别处理 num=0 的情况
    if(!length) {
        numStr[0] = '0';
        ++length;
    }

    // 将字符串倒转，使得 numStr[0]保存的是 num 的高位数字
    for(int i = 0, j = length - 1; i < j; ++i, --j) {
        swap(numStr[i], numStr[j]);
    }

    numStr[length] = '\0';
}
```

3）实现中断处理函数。

```
extern "C" void setup_kernel()
{
    // 中断处理部件
    interruptManager.initialize();
    // 屏幕 IO 处理部件
    stdio.initialize();
    interruptManager.enableTimeInterrupt();
    interruptManager.setTimeInterrupt((void
*)asm_time_interrupt_handler);
    //asm_enable_interrupt();
    printf("print percentage: %%\n"
           "print char \"N\": %c\n"
           "print string \"Hello World!\": %s\n"
           "print decimal: \"-1234\": %d\n"
           "print hexadecimal \"0x7abcdef0\": %x\n"
           "print float: \"6.66666666\":%f\n"
           "print .float: \"6.66666666\":%.3f\n",
           'N', "Hello World!", -1234, 0x7abcdef0,6.66666666,6.66666666);
    //uint a = 1 / 0;
    asm_halt();
}
```

- 实验结果展示：

---------- 实验任务 2：线程的实现 ----------

- 任务要求： 自行设计 PCB，可以添加更多的属性，如优先级等，然后根据你的 PCB 来实现线程，演示执行结果。
- 思路分析： 本实验设计了一个简单的线程管理系统，基于 PCB 和 时间片轮转调度算法实现多线程调度：

1.PCB 设计：包含线程栈、状态、优先级、PID、执行时间等属性，用于管理线程的执行环境。

2.线程创建：通过 executeThread 分配 PCB，初始化线程栈（存储函数地址、参数、返回地址等），并加入就绪队列。

3.线程调度：采用 RR 算法，每个线程运行固定时间片（ticks = priority * 10），时间片耗尽后切换线程。

4.上下文切换：通过汇编代码 asm_switch_thread 保存/恢复寄存器，实现线程栈切换。

- 实验步骤：

1.描述线程。构建结构体 PCB，包括成员变量线程栈、五个状态、优先级、运行时间、线程负责运行的函数和函数的参数等。

1）stack 是每一个线程独立拥有的，栈保存在线程 PCB 中。Struct PCB 的地址是分配的页的低地址，线程栈指针起止位置为页的最高地址，栈的扩展方向由高到低。

2）status 是线程的状态，如运行态、阻塞态和就绪态等。

3）name 是线程的名称。

4）priority 是线程的优先级，线程的优先级决定了抢占式调度的过程和线程的执行时间。

5）pid 是线程的标识符，每一个线程的 pid 都是唯一的。

6）ticks 是线程剩余的执行次数。在时间片调度算法中，每发生中断一次记为一个 tick，当 ticks=0 时，线程会被换下处理器，然后将其他线程换上处理器执行。

7）ticksPassedBy 是线程总共执行的 tick 的次数。

8）tagInGeneralList 和 tagInAllList 是线程在线程队列中的标识，用于在线程队列中找到线程的 PCB。

9）声明程序管理类 ProgramManager 用于进程创建和管理。

```
#ifndef PROGRAM_H
#define PROGRAM_H

class ProgramManager
{

};

#endif
```

2.PCB 的分配

1）在创建线程之前，我们需要向内存申请一个 PCB。我们将一个 PCB 的大小设置为 4096 个字节，也就是一个页的大小。目前我们在内存中预留若干个 PCB 的内存空间来存放和管理 PCB。

```
// PCB 的大小，4KB。
const int PCB_SIZE = 4096;
// 存放 PCB 的数组，预留了 MAX_PROGRAM_AMOUNT 个 PCB 的大小空间。
char PCB_SET[PCB_SIZE * MAX_PROGRAM_AMOUNT];
// PCB 的分配状态，true 表示已经分配，false 表示未分配。
```

```
bool PCB_SET_STATUS[MAX_PROGRAM_AMOUNT];
```

2）在 ProgramManager 中声明两个管理 PCB 所在的内存空间函数。

```
// 分配一个 PCB
PCB *allocatePCB()
{
    for (int i = 0; i < MAX_PROGRAM_AMOUNT; ++i)
    {
        if (!PCB_SET_STATUS[i])
        {
            PCB_SET_STATUS[i] = true;
            return (PCB *)((int)PCB_SET + PCB_SIZE * i);
        }
    }

    return nullptr;
}

// 归还一个 PCB
void releasePCB(PCB *program)
{
    int index = ((int)program - (int)PCB_SET) / PCB_SIZE;
    PCB_SET_STATUS[index] = false;
}
```

allocatePCB 会去检查 PCB_SET 中每一个 PCB 的状态，如果找到一个未被分配的 PCB，则返回这个 PCB 的起始地址。因为 PCB_SET 中的 PCB 是连续存放的，对于第 i 个 PCB，PCB_SET 的首地址加上 i×PCB_SIZE 就是第 i 个 PCB 的起始地址。PCB 的状态保存在 PCB_SET_STATUS 中，并且 PCB_SET_STATUS 的每一项会在 ProgramManager 总被初始化为 false，表示所有的 PCB 都未被分配。被分配的 PCB 用 true 来标识。

如果 PCB_SET_STATUS 的所有元素都是 true，表示所有的 PCB 都已经被分配，此时应该返回 nullptr，表示 PCB 分配失败。

releasePCB 接受一个 PCB 指针 program，然后计算出 program 指向的 PCB 在 PCB_SET 中的位置，然后将 PCB_SET_STATUS 中的对应位置设置 false 即可。

3.线程的创建。

1）先在 ProgramManager 中放入两个 List 成员，allPrograms 和 readyPrograms。

```
class ProgramManager
{
public:
```

```
    List allPrograms;   // 所有状态的线程/进程的队列
    List readyPrograms; // 处于 ready(就绪态)的线程/进程的队列

public:
    ProgramManager();
    void initialize();

    // 分配一个 PCB
    PCB *allocatePCB();
    // 归还一个 PCB
    // program: 待释放的 PCB
    void releasePCB(PCB *program);
};
```

allPrograms 是所有状态的线程和进程的队列，其中放置的是的 PCB::tagInAllList。readyPrograms 是处在 ready(就绪态)的线程/进程的队列，放置的是 PCB::tagInGeneralList。

2）使用 ProgramManager 的成员函数前，我们必须初始化 ProgramManager。

```
ProgramManager::ProgramManager()
{
    initialize();
}

void ProgramManager::initialize()
{
    allPrograms.initialize();
    readyPrograms.initialize();
    running = nullptr;

    for (int i = 0; i < MAX_PROGRAM_AMOUNT; ++i)
    {
        PCB_SET_STATUS[i] = false;
    }
}
```

3）开始创建线程。线程实际上执行的是某一个函数的代码。但是，并不是所有的函数都可以放入到线程中执行的。这里我们规定线程只能执行返回值为 void，参数为 void *的函数，其中，void *指向了函数的参数。我们把这个函数定义为 ThreadFunction。线程只能执行如此函数 **typedef void**(*ThreadFunction)(**void** *).

4) 在 ProgramManager 中声明一个用于创建线程的函数 executeThread。

```
class ProgramManager
{
public:
    List allPrograms;   // 所有状态的线程/进程的队列
    List readyPrograms; // 处于 ready(就绪态)的线程/进程的队列
```

```
    PCB *running;       // 当前执行的线程
public:
    ProgramManager();
    void initialize();

    // 创建一个线程并放入就绪队列
    // function：线程执行的函数
    // parameter：指向函数的参数的指针
    // name：线程的名称
    // priority：线程的优先级
    // 成功，返回 pid；失败，返回-1
    int executeThread(ThreadFunction function, void *parameter, const
char *name, int priority);

    // 分配一个 PCB
    PCB *allocatePCB();
    // 归还一个 PCB
    // program：待释放的 PCB
    void releasePCB(PCB *program);
};
```

5) 实现 executeThread.。

```
int    ProgramManager::executeThread(ThreadFunction    function,    void
*parameter, const char *name, int priority)
{
    // 关中断，防止创建线程的过程被打断
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    // 分配一页作为 PCB
    PCB *thread = allocatePCB();

    if (!thread)
        return -1;

    // 初始化分配的页
    memset(thread, 0, PCB_SIZE);

    for (int i = 0; i < MAX_PROGRAM_NAME && name[i]; ++i)
    {
        thread->name[i] = name[i];
    }

    thread->status = ProgramStatus::READY;
    thread->priority = priority;
    thread->ticks = priority * 10;
    thread->ticksPassedBy = 0;
    thread->pid = ((int)thread - (int)PCB_SET) / PCB_SIZE;

    // 线程栈
    thread->stack = (int *)((int)thread + PCB_SIZE);
    thread->stack -= 7;
    thread->stack[0] = 0;
    thread->stack[1] = 0;
    thread->stack[2] = 0;
    thread->stack[3] = 0;
```

```
    thread->stack[4] = (int)function;
    thread->stack[5] = (int)program_exit;
    thread->stack[6] = (int)parameter;

    allPrograms.push_back(&(thread->tagInAllList));
    readyPrograms.push_back(&(thread->tagInGeneralList));

    // 恢复中断
    interruptManager.setInterruptStatus(status);

    return thread->pid;
}
```

分析线程创建逻辑。

i. 实现线程互斥。多线程环境下 PCB 的分配工作需要线程互斥处理，这里只使用开关中断来实现互斥。我们在时钟中断发生时进行线程调度，关中断后时钟中断无法被响应因而线程无法被调度直到开中断，线程可以安全创建不被打断。

```
class InterruptManager
{

    ...

    // 开中断
    void enableInterrupt();
    // 关中断
    void disableInterrupt();
    // 获取中断状态
    // 返回 true, 中断开启; 返回 false, 中断关闭
    bool getInterruptStatus();
    // 设置中断状态
    // status=true, 开中断; status=false, 关中断
    void setInterruptStatus(bool status);

    ...
};
```

ii. 第 8 行，关中断后，我们向 PCB_SET 申请一个线程的 PCB，然后我们在第 14 行使用 memeset 将 PCB 清 0。第 16-25 行，我们设置 PCB 的成员 name、status、priority、ticks、ticksPassedBy 和 pid。这里，线程初始的 ticks 我们简单地设置为 10 倍的 priority。pid 则简单地使用 PCB 在 PCB_SET 的位置来代替。第 28 行，我们初始化线程的栈。我们将栈放置在 PCB 中，而线程的栈是从 PCB 的顶部开始向下增长的，所以不会与位于 PCB 低地址的 name 和 pid 等变量冲突。线程栈的初始地址是 PCB 的起始地址加上 PCB_SIZE。第 29-36 行，我们在栈中放入 7 个整数值。4 个为 0 的值是要放到 ebp，ebx，edi，esi 中的。thread->stack[4]是线程执行的函数的起始地址。thread->stack[5]是线程的返

回地址，所有的线程执行完毕后都会返回到这个地址。thread->stack[6]是线程的参数的地址。

创建完线程的 PCB 后，我们将其放入到 allPrograms 和 readyPrograms 中，等待时钟中断来的时候，这个新创建的线程就可以被调度上处理器。

最后我们将中断的状态恢复，此时我们便创建了一个线程。

4.线程的调度。

1）先在 ProgramManager 中放入成员 running，表示当前在处理机上执行的线程的 PCB。

```
class ProgramManager
{
public:
    List allPrograms;   // 所有状态的线程/进程的队列
    List readyPrograms; // 处于 ready(就绪态)的线程/进程的队列
    PCB *running;       // 当前执行的线程

    ...
};
```

2）修改之前的处理时钟中断函数。

```
extern "C" void c_time_interrupt_handler()
{
    PCB *cur = programManager.running;

    if (cur->ticks)
    {
        --cur->ticks;
        ++cur->ticksPassedBy;
    }
    else
    {
        programManager.schedule();
    }
}
```

3）实现线程调度算法时间片轮转算法（Round Robin, RR）。当时钟中断到来时，我们对当前线程的 ticks 减 1，直到 ticks 等于 0，然后执行线程调度。线程调度的是通过函数 ProgramManager::schedule 来完成的。

```
void ProgramManager::schedule()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    if (readyPrograms.size() == 0)
    {
        interruptManager.setInterruptStatus(status);
        return;
```

```
    }

    if (running->status == ProgramStatus::RUNNING)
    {
        running->status = ProgramStatus::READY;
        running->ticks = running->priority * 10;
        readyPrograms.push_back(&(running->tagInGeneralList));
    }
    else if (running->status == ProgramStatus::DEAD)
    {
        releasePCB(running);
    }

    ListItem *item = readyPrograms.front();
    PCB *next = ListItem2PCB(item, tagInGeneralList);
    PCB *cur = running;
    next->status = ProgramStatus::RUNNING;
    running = next;
    readyPrograms.pop_front();

    asm_switch_thread(cur, next);

    interruptManager.setInterruptStatus(status);
}
```

分析 ProgramManager::schedule 的逻辑。

首先，和 ProgramManager::executeThread 一样，为了实现线程互斥，在进程线程调度前，我们需要关中断，退出时再恢复中断。

第 6-9 行，我们判断当前可调度的线程数量，如果 readyProgram 为空，那么说明当前系统中只有一个线程，因此无需进行调度，直接返回即可。

第 12-21 行，我们判断当前线程的状态，如果是运行态(RUNNING)，则重新初始化其状态为就绪态(READY)和 ticks，并放入就绪队列；如果是终止态(DEAD)，则回收线程的 PCB。

第 23 行，我们去就绪队列的第一个线程作为下一个执行的线程。就绪队列的第一个元素是 ListItem *类型的，我们需要将其转换为 PCB。注意到放入就绪队列 readyPrograms 的是每一个 PCB 的&tagInGeneralList，而 tagInGeneralList 在 PCB 中的偏移地址是固定的。也就是说，我们将 item 的值减去 tagInGeneralList 在 PCB 中的偏移地址就能够得到 PCB 的起始地址。我们将上述过程写成一个宏。

```
    #define ListItem2PCB(ADDRESS, LIST_ITEM) ((PCB *)((int)(ADDRESS) -
(int)&((PCB *)0)->LIST_ITEM))
```

其中，(int)&((PCB *)0)->LIST_ITEM 求出的是 LIST_ITEM 这个属性在 PCB

中的偏移地址。

第 27-28 行，我们从就绪队列中删去第一个线程，设置其状态为运行态和当前正在执行的线程。

最后，我们就开始将线程从 cur 切换到 next。线程的所有信息都在线程栈中，只要我们切换线程栈就能够实现线程的切换，线程栈的切换实际上就是将线程的栈指针放到 esp 中。

```
asm_switch_thread:
    push ebp
    push ebx
    push edi
    push esi

    mov eax, [esp + 5 * 4]
    mov [eax], esp ; 保存当前栈指针到 PCB 中，以便日后恢复

    mov eax, [esp + 6 * 4]
    mov esp, [eax] ; 此时栈已经从 cur 栈切换到 next 栈

    pop esi
    pop edi
    pop ebx
    pop ebp

    sti
    ret
```

第 2-5 行，我们保存寄存器 ebp，ebx，edi，esi。为什么要保存这几个寄存器？这是由 C 语言的规则决定的，C 语言要求被调函数主动为主调函数保存这 4 个寄存器的值。如果我们不遵循这个规则，那么当我们后面线程切换到 C 语言编写的代码时就会出错。

第 7-8 行，我们保存 esp 的值到线程的 PCB::stack 中，用做下次恢复。注意到 PCB::stack 在 PCB 的偏移地址是 0。因此，第 7 行代码是首先将 cur->stack 的地址放到 eax 中，第 8 行向[eax]中写入 esp 的值，也就是向 cur->stack 中写入 esp。

第 10-11 行，我们将 next->stack 的值写入到 esp 中，从而完成线程栈的切换。

接下来的 pop 语句会将 4 个 0 值放到 esi，edi，ebx，ebp 中。此时，栈顶的数据是线程需要执行的函数的地址 function。执行 ret 返回后，function 会被加载进 eip，从而使得 CPU 跳转到这个函数中执行。此时，进入函数后，函数的栈顶是函数的返回地址，返回地址之上是函数的参数，符合函数的调用规则。而函数

执行完成时，其执行 ret 指令后会跳转到返回地址 program_exit。

```
void program_exit()
{
    PCB *thread = programManager.running;
    thread->status = ThreadStatus::DEAD;

    if (thread->pid)
    {
        programManager.schedule();
    }
    else
    {
        interruptManager.disableInterrupt();
        printf("halt\n");
        asm_halt();
    }
}
```

program_exit 会将返回的线程的状态置为 DEAD，然后调度下一个可执行的线程上处理器。注意，我们规定第一个线程是不可以返回的，这个线程的 pid 为 0。

执行 4 个 pop 后，之前保存在线程栈中的内容会被恢复到这 4 个寄存器中，然后执行 ret 后会返回调用 asm_switch_thread 的函数，也就是 ProgramManager::schedule，然后在 ProgramManager::schedule 中恢复中断状态，返回到时钟中断处理函数，最后从时钟中断中返回，恢复到线程被中断的地方继续执行。

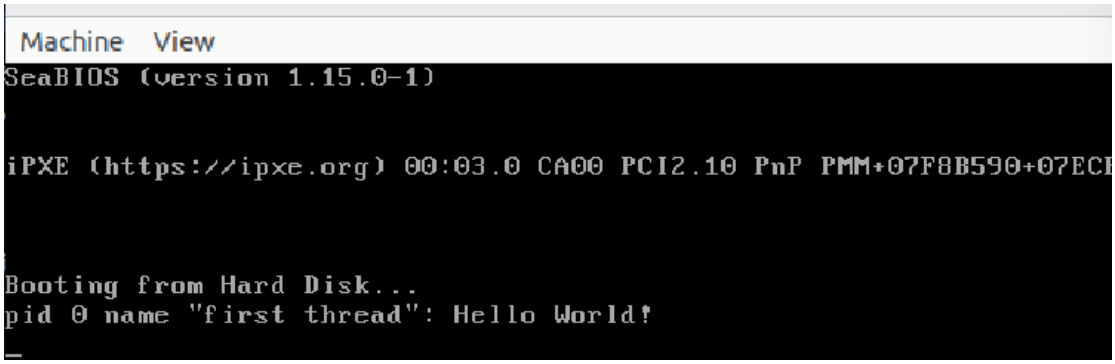
这样，通过 asm_switch_thread 中的 ret 指令和 esp 的变化，我们便实现了线程的调度。

5.创建第一个线程并输出“Hello World”，pid 和线程的 name。

```
#include "asm_utils.h"
#include "interrupt.h"
#include "stdio.h"
#include "program.h"
#include "thread.h"

// 屏幕 IO 处理器
STDIO stdio;
// 中断管理器
InterruptManager interruptManager;
// 程序管理器
ProgramManager programManager;

void first_thread(void *arg)
{
    // 第 1 个线程不可以返回
```

```
    printf("pid %d name \"%s\": Hello World!\n",
           programManager.running->pid, programManager.running->name);
    asm_halt();
}

extern "C" void setup_kernel()
{

    // 中断管理器
    interruptManager.initialize();
    interruptManager.enableTimeInterrupt();
    interruptManager.setTimeInterrupt((void
*)asm_time_interrupt_handler);

    // 输出管理器
    stdio.initialize();

    // 进程/线程管理器
    programManager.initialize();

    // 创建第一个线程
    int pid = programManager.executeThread(first_thread, nullptr, "first
thread", 1);
    if (pid == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }

    ListItem *item = programManager.readyPrograms.front();
    PCB *firstThread = ListItem2PCB(item, tagInGeneralList);
    firstThread->status = RUNNING;
    programManager.readyPrograms.pop_front();
    programManager.running = firstThread;
    asm_switch_thread(0, firstThread);

    asm_halt();
}
```

- 实验结果展示：



---------- 实验任务 3 ： 线程调度切换的秘密----------

- 任务要求：编写若干个线程函数，使用 gdb 跟踪 c_time_interrupt_handler、

asm_switch_thread 等函数，观察线程切换前后栈、寄存器、PC 等变化，结合 gdb、材料中“线程的调度”的内容来跟踪并说明下面两个过程。

1.一个新创建的线程是如何被调度然后开始执行的。

2.一个正在执行的线程是如何被中断然后被换下处理器的，以及换上处理机后又是如何从被中断点开始执行的。

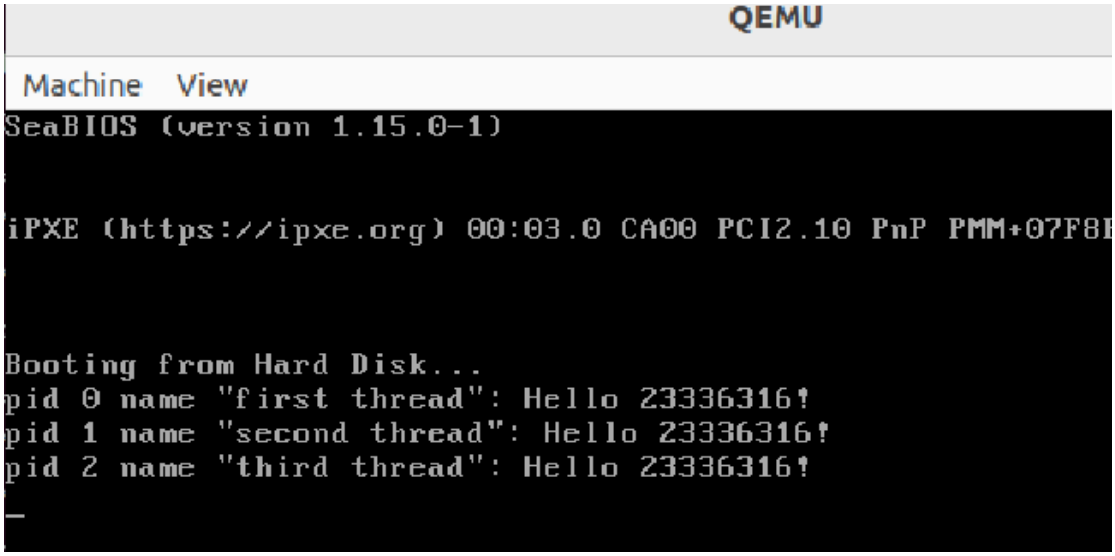
- 思路分析：本实验通过创建多个线程，结合 GDB 调试跟踪 c_time_interrupt_handler 和 asm_switch_thread，观察线程切换时的栈、寄存器、PC 变化，分析新线程如何被调度执行以及运行中线程如何被中断并恢复执行，验证时间片轮转调度机制。

- 实验步骤：

1．添加两个线程。

```
void third_thread(void *arg) {
    printf("pid     %d     name     \"%s\":     Hello     23336316!\n",
programManager.running-&gt;pid, programManager.running-&gt;name);
  //  while(1) {

    //}
       // programManager.schedule();
}
void second_thread(void *arg) {
    printf("pid     %d     name     \"%s\":     Hello     23336316!\n",
programManager.running-&gt;pid, programManager.running-&gt;name);
      //  programManager.schedule();
}
void first_thread(void *arg)//创建第一个线程
{
    // 第 1 个线程不可以返回
    printf("pid     %d     name     \"%s\":     Hello     23336316!\n",
programManager.running-&gt;pid, programManager.running-&gt;name);
    if (!programManager.running-&gt;pid)
    {
        programManager.executeThread(second_thread,    nullptr,    "second
thread", 1);//创建第二个线程
        programManager.executeThread(third_thread,    nullptr,    "third
thread", 1);//创建第三个线程
      //  programManager.schedule();
    }
    asm_halt();
}
```

2.运行和调试。



屏幕上相继出现第一二三行语句。

gdb 调试。

创建断点。

```
(gdb) b first_thread
Breakpoint 1 at 0x2079d: file ../src/kernel/setup.cpp, line 26.
(gdb) b second_thread
Breakpoint 2 at 0x20772: file ../src/kernel/setup.cpp, line 21.
(gdb) b third_thread
Breakpoint 3 at 0x20747: file ../src/kernel/setup.cpp, line 14.
```

```
(gdb) b executeThread
Breakpoint 4 at 0x2037c: file ../src/kernel/program.cpp, line 31.
(gdb) b asm_switch_thread
Breakpoint 5 at 0x2148c: file ../src/utils/asm_utils.asm, line 24.
(gdb) b c_time_interrupt_handler
```

```
asm_switch_thread:
    push ebp
    push ebx
    push edi
    push esi

    mov eax, [esp + 5 * 4]
    mov [eax], esp ; 保存当前栈指针到 PCB 中，以便日后恢复

    mov eax, [esp + 6 * 4]
    mov esp, [eax] ; 此时栈已经从 cur 栈切换到 next 栈
```

```
    pop esi
    pop edi
    pop ebx
    pop ebp

    sti
    ret
```

1）第一个断点打在第一个线程创建时的函数 executeThread 处，此时属于进程创建前。看看线程创建前寄存器、栈、pc 情况。

```
      30  int ProgramManager::executeThread(ThreadFunction function, void *pa
B+>   31  {
      32      // 关中断，防止创建线程的过程被打断
      33      bool status = interruptManager.getInterruptStatus();
      34      interruptManager.disableInterrupt();
```

```
#0  ProgramManager::executeThread (this=0x31d40 <programManager>,
    function=0x2079d <first_thread(void*)>, parameter=0x0,
    name=0x215da "first thread", priority=1) at ../src/kernel/program.cpp:3
```

当前正在进行线程的创建。

```
(gdb) info register ebp ebx edi esi eip eax
ebp            0x7bfc             0x7bfc
ebx            0x39000            233472
edi            0x0                0
esi            0x0                0
eip            0x2037c            0x2037c <ProgramManager::executeThread(void (
*)(void*), void*, char const*, int)>
eax            0x31d2f            204079
```

这是几个寄存器的情况。↑

```
(gdb)  x/20x $esp
0x7bc0: 0x0002088b      0x00031d40      0x0002079d      0x00000000
0x7bd0: 0x000215da      0x00000001      0x000214b0      0x00000000
0x7be0: 0x00000080      0x00000001      0x00038e00      0x000000cd
0x7bf0: 0x00000000      0x00007eab      0x00038e00      0x00000000
0x7c00: 0xd88ec031      0xc08ed08e      0xe88ee08e      0xb87c00bc
```

这是栈的情况，栈中保存了线程的初始化数据。↑ 内容：0x0002088b（setup_kernel 调用后的下一条指令）。0x0002079d：线程函

数 first_thread 的地址。0x000215da：线程名称字符串 "first thread" 的地址。0x00000001：优先级参数。

executeThread 完成后，会将线程加入就绪队列，后续通过 asm_switch_thread 切换到该线程。

2）当第一个线程创建完。

```
ebp            0x7bbc              0x7bbc
ebx            0x39000             233472
edi            0x0                 0
esi            0x0                 0
eip            0x204ce             0x204ce <ProgramManager::executeThread(void (
*)(void*), void*, char const*, int)+338>
eax            0x22d18             142616
```

一直找不到 first_thread 刚创建出来的状态，那这个吧。

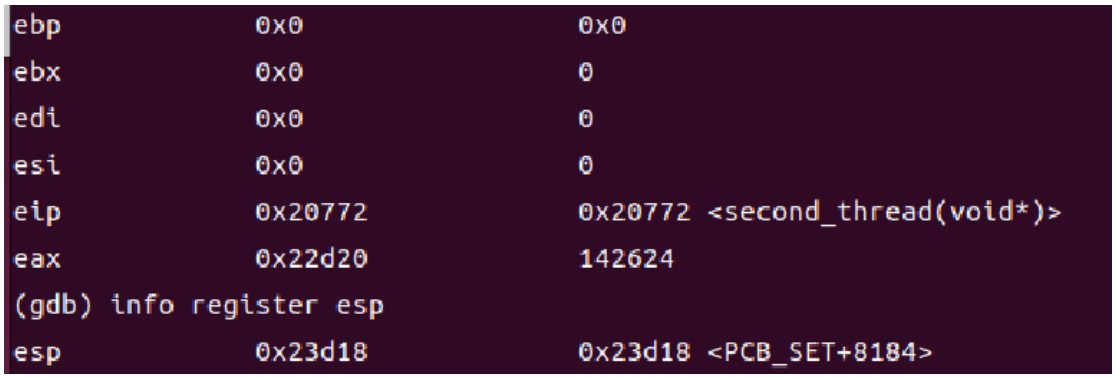
```
(gdb) bt
#0  ProgramManager::executeThread (this=0x31d40 <programManager>,
    function=0x2079d <first_thread(void*)>, parameter=0x0,
    name=0x215da "first thread", priority=1) at ../src/kernel/program.cpp:65
#1  0x0002088b in setup_kernel () at ../src/kernel/setup.cpp:53
#2  0xd88ec031 in ?? ()
```

```
(gdb) x/20x $esp
0x7ba4: 0x00031d48      0x00021d20      0x000214e4      0x0000000c
0x7bb4: 0x00007bcc      0x0002023e      0x00007bfc      0x0002088b
0x7bc4: 0x00031d40      0x0002079d      0x00000000      0x000215da
0x7bd4: 0x00000001      0x000214b0      0x00000000      0x00000080
0x7be4: 0x00000001      0x00038e00      0x000000cd      0x00000000
```

```
(gdb) info register esp
esp            0x7ba4              0x7ba4
```

可以发现 esp 指向栈顶。

3）切换到第二个进程。



可以发现 ebp,ebx,edi,esi 的值都为零，esp 存储的是栈顶的地址。

总结：

1.新线程如何被调度并开始执行？

1）线程创建：在 setup_kernel() 中调用 programManager.executeThread()：

i. 分配 PCB（进程控制块）和线程栈。

ii. 初始化栈帧：伪造中断返回现场（保存 eip 指向线程入口函数 first_thread）。

iii. 设置线程状态为 READY 并加入就绪队列 readyPrograms

2) 触发调度

i.首次通过 asm_switch_thread(0, firstThread) 强制切换到新线程（调试信息中 eip=0x2037c 为 executeThread 的地址）。

ii. 后续由时钟中断（c_time_interrupt_handler）触发调度。

3)上下文加载

i. asm_switch_thread 执行以下操作：

```
pushad              ; 保存当前寄存器到旧线程栈
mov esp, [next->esp] ; 切换到新线程栈
popad               ; 从新线程栈恢复寄存器
ret                 ; eip 跳转到新线程入口
```

ii. 关键寄存器变化：eip 指向 first_thread，esp 切换到新进程栈顶。

4) .线程开始执行：cpu 从 first_thread 第一条指令开始执行，线程开始正式运行。

2. **正在执行的线程如何被中断并切换？**

1）时钟中断发生时 cpu 自动保存当前 **cs:eip** 和 **eflags** 到内核栈，跳转到 **c_time_interrupt_handler**。若当前线程时间片耗尽（ticks == 0），调用 programManager.schedule()。

2）调度器工作

i 保存当前线程。将运行中的线程从 running 改成 ready，重新计算时间片。将其 PCB 放回就绪队列，把所有的寄存器压入旧线程栈。

ii 选择新的线程。从就绪队列中按照调度算法选择新的线程，把状态设置成 running，从新线程栈恢复寄存器。

---------- 实验任务 4 ： 修改调度算法----------

- 任务要求：实现自己的调度算法。以下实现先来先服务算法。
- 思路分析： 本实验实现 先来先服务调度算法：

  1.移除时间片变量，取消时间片轮转调度逻辑。

  2.按就绪队列顺序调度，线程结束时释放 PCB，若无线程则系统停机。

  3.修改线程退出逻辑，确保所有线程（包括 PID=0）触发调度。

  4.创建 3 个测试线程，依次执行并输出信息，验证 FCFS 调度顺序。
- 实验步骤：

  1.把 PCB 中关于时间调度的变量注释掉。

```
struct PCB
{
    int *stack;                         // 栈指针，用于调度时保存esp
    char name[MAX_PROGRAM_NAME + 1]; // 线程名
    enum ProgramStatus status;          // 线程的状态
    int priority;                       // 线程优先级
    int pid;                            // 线程pid
//    int ticks;                          // 线程时间片总时间
//    int ticksPassedBy;                  // 线程已执行时间
    ListItem tagInGeneralList;          // 线程队列标识
    ListItem tagInAllList;              // 线程队列标识
};
    thread->priority = priority;
//    thread->ticks = priority * 10;
//    thread->ticksPassedBy = 0;
```

2.修改调度函数，当前线程结束时输出“Current thread exists!”，释放 PCB，running 指向空；当前没有线程运行且就绪队列没有新线程时，打印“No thread！Halting！”；当没有线程运行并且就绪队列不为空，按照线程加入队列的顺序进行先来先服务的调度，输出“Begin scheduling！”。

```
void ProgramManager::schedule(){
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    if (!running && readyPrograms.size() > 0)//开始调度
    {
        printf("Begin scheduling!\n");

    }
    if (!running && readyPrograms.size() == 0)//当前没有线程并且没有新的线程
    {
        printf("No thread! Halting!\n");
        asm_halt();
        return;
    }
    if (running&&running->status == ProgramStatus::DEAD)//当前线程结束
    {
        releasePCB(running);
        printf("Current thread exits!\n");
        running=nullptr;
        return;
    }

    ListItem *item = readyPrograms.front();
    PCB *next = ListItem2PCB(item, tagInGeneralList);
    next->status = ProgramStatus::RUNNING;
    running = next;
    readyPrograms.pop_front();

    asm_switch_thread(0, next);

    interruptManager.setInterruptStatus(status);
}
```

3.删去中断处理函数原有逻辑。

```
// 中断处理函数
extern "C" void c_time_interrupt_handler()
{

}
```

4.修改退出函数，把原来的 if（pid）改成 if（1），避免第一个线程 pid 为 0 导致不会进行后面的调度。

```
void program_exit()
{
    PCB *thread = programManager.running;
    thread->status = ProgramStatus::DEAD;

    if (1)
    {
        programManager.schedule();
    }
    else
    {
        interruptManager.disableInterrupt();
        printf("halt\n");
        asm_halt();
    }
}
```

4.写三个线程。

```
void third_thread(void *arg) {
    printf("pid %d name \"%s\": Hello 23336316!\n", programManager.running->pid, programManager.running->name);
    program_exit();
}
void second_thread(void *arg) {
    printf("pid %d name \"%s\": Hello 23336316!\n", programManager.running->pid, programManager.running->name);
    program_exit();
}
void first_thread(void *arg)
{
    // 第1个线程不可以返回
    printf("pid %d name \"%s\": Hello 23336316!\n", programManager.running->pid, programManager.running->name);
    if (!programManager.running->pid)
    {
        programManager.executeThread(second_thread, nullptr, "second thread", 1);
        programManager.executeThread(third_thread, nullptr, "third thread", 1);
    }
    program_exit();
}
```

在当一个线程中创建第二个和第三个线程，线程按顺序加入了就绪队列。每一个线程运行时输出 pid，name 和 “Hello 23336316！”线程输出后主动退出开始调度。

- 实验结果展示：通过执行前述代码，可得下图结果。

```
Booting from Hard Disk...
pid 0 name "first thread": Hello 23336316!
Current thread exits!
Begin scheduling!
pid 1 name "second thread": Hello 23336316!
Current thread exits!
Begin scheduling!
pid 2 name "third thread": Hello 23336316!
Current thread exits!
No thread! Halting!
```

可以从上图看出调度过程。先是第一个线程运行输出；随后线程退出结束，打印“Current thread exits！”；最后开始调度，打印“Begin scheduling！”。第二个线程同理。到第三个线程退出时，调度算法发现当前没有运行线程并且就绪队列为空，打印“No thread！ Halting！”后停止。

思考题：

1. 定义两个宏实现 a++和++a 的逻辑。va_after_add 实现 a++的逻辑，a+=1 自增，随后-1 得到指向原来的 a 的指针，随后解指针得到原来的 a 值。va_before_add 实现++a 的逻辑，解指针得到的是自增后的 a 值。

```
#define va_after_add(a)(*(int*)((a+=1)-1)) //实现a++的逻辑，a=a+1,之后得到即原来的a
#define va_before_add(a)(*(int*)(a+=1))//实现++a逻辑，a=a+1,得到加后的a
```

```
else
{
    va_before_add(row);
}
```

```
int np = fmt[va_before_add(i)]-'0';
```

```
va_after_add(i);
```

将 stdio.cpp 的所有自增全改了，发现最终输出和原来是一样的。

3. 如何将一个正整数转换为任意进制对应的字符串：

我们利用函数 itos 来进行转换。itos 函数的作用是将一个无符号 32 位整数 num 转换为指定进制 mod 的字符串表示，并将结果存储在 numStr 字符数组中。

length 用于记录字符串长度。通过不断循环来对 num 取模和除法，得到当

前最低位数字，然后将数字转化为字符进行存储。如果最低位数字大于 9，转化为大写字母。最后翻转字符，因为转换过来的字符串是逆序的。在 numStr 的末尾添加“\0”成为合法的字符串。

Section 5 实验总结与心得体会

本次实验也是收获满满。我学习了可变参数的原理，手动实现了自己的 printf；我自行设计了一个 PCB，通过 gdb 来测试，观察栈和寄存器的变化，更加深入了对线程运行的理解；我修改了原有的调度算法使之实现了先来先服务的原则，正确运行三个程序。总之，这一次实验加深了我对线程的理解，也收获了线程成功运行、调度算法成功实现的满满的成就感。